EECS 370

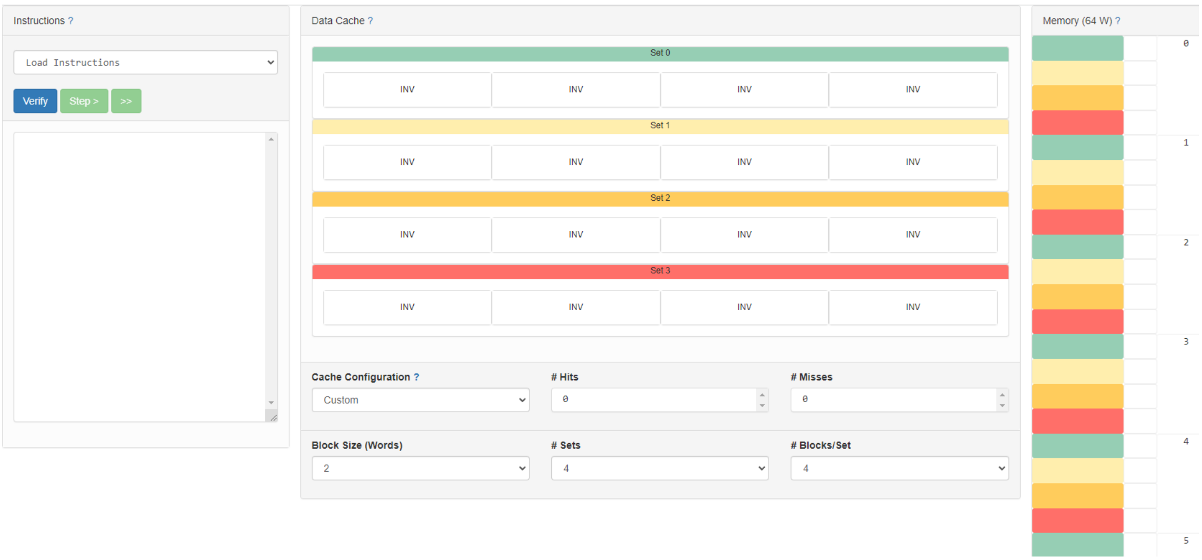
Virtual Memory Basics

Announcements

- P4
  - Last project!
  - Due Thur (11/30)
- HW 4
  - Last homework!
  - Due Mon (12/4)
- Final exam
  - …Last exam!
  - Tue (12/12) @ 10:30 am

P4

- Check out simulator on website (not same as project... doesn't cache instrs)



Practice Problem 4: Guess that cache!

Similar to homework! Here is the series of address references (in hex) to a cache of size 512 bytes. You are asked to **determine the configuration of the cache**. Assume 12-bit addresses

```
0x310 – Miss
0x30f – Miss
0x510 – Miss
0x31f – Hit
0x72d – Miss
0x72f – Hit
0x320 – Miss
0x520 – Miss
0x720 - Miss
```

Practice Problem 4: Guess that cache!

Here is the series of address references (in hex) to a cache of size 512 bytes. You are asked to **determine the configuration of the cache**. Assume 12-bit addresses

0x310 – Miss
0x30f – Miss
0x510 – Miss
0x31f – Hit
0x72d – Miss
0x72f – Hit
0x320 – Miss
0x520 – Miss
0x720 - Miss

<u>Determine block size</u>

First hit must be brought in by another miss

Take closest address: 0x310, so know block size must be at least 16 bytes so 0x31f brought in when 0x310 miss occurs

Now, is the block size larger? Know that 0x30f was a miss, thus 0x310 and 0x30f not in the same block. Thus, block size must be <= 16 bytes

Thus Block Size = 16 bytes

Practice Problem 4: Guess that cache!

Here is the series of address references (in hex) to a cache of size 512 bytes. You are asked to **determine the configuration of the cache**. Assume 12-bit addresses

0x310 – Miss
0x30f – Miss
0x510 – Miss
0x31f – Hit
0x72d – Miss
0x72f – Hit
0x320 – Miss
0x520 – Miss
0x720 - Miss

<u>Determine associativity</u>

Assume direct mapped: 3-bit tag, 5-bit index, 4-bit offset.
If DM, 0x310 and 0x510 would both map to index 17,
Thus 0x31f could not be a hit. So, not direct mapped.

Assume 2-way associative: 4-bit tag, 4-bit index, 4-bit offset
This fixes the green accesses, and allows 0x31f to be a hit.

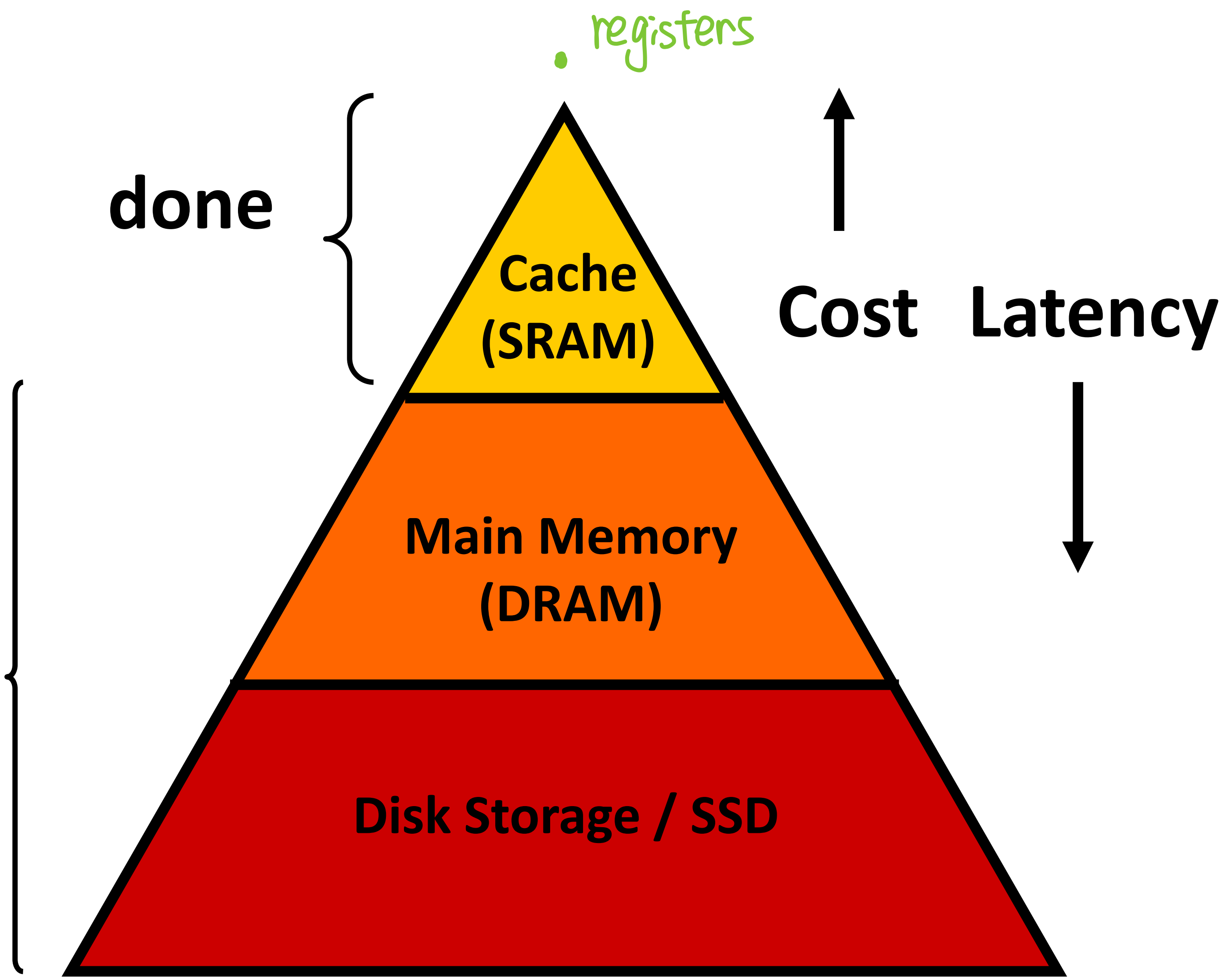
What about > 2-way associative?
Now we also know that 0x720 is a miss even though 3 accesses earlier 0x72f was a hit, and thus it is in the cache. The intervening 2 accesses must kick it out, 0x320 and 0x520. Both go to set 2. If the associativity was > 2, then 0x720 would be a hit. So, must conclude that cache is 2-way associative.

Lastly, <u>number of sets = 512 / (2 * 16) = 16</u>

Agenda

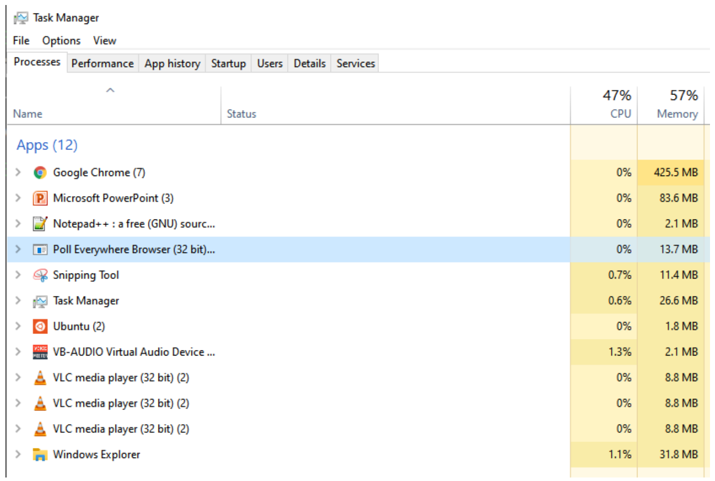
- **Motivation**
- Virtual Memory Principles
- Page Tables
- Class Problem

Storage Hierarchy



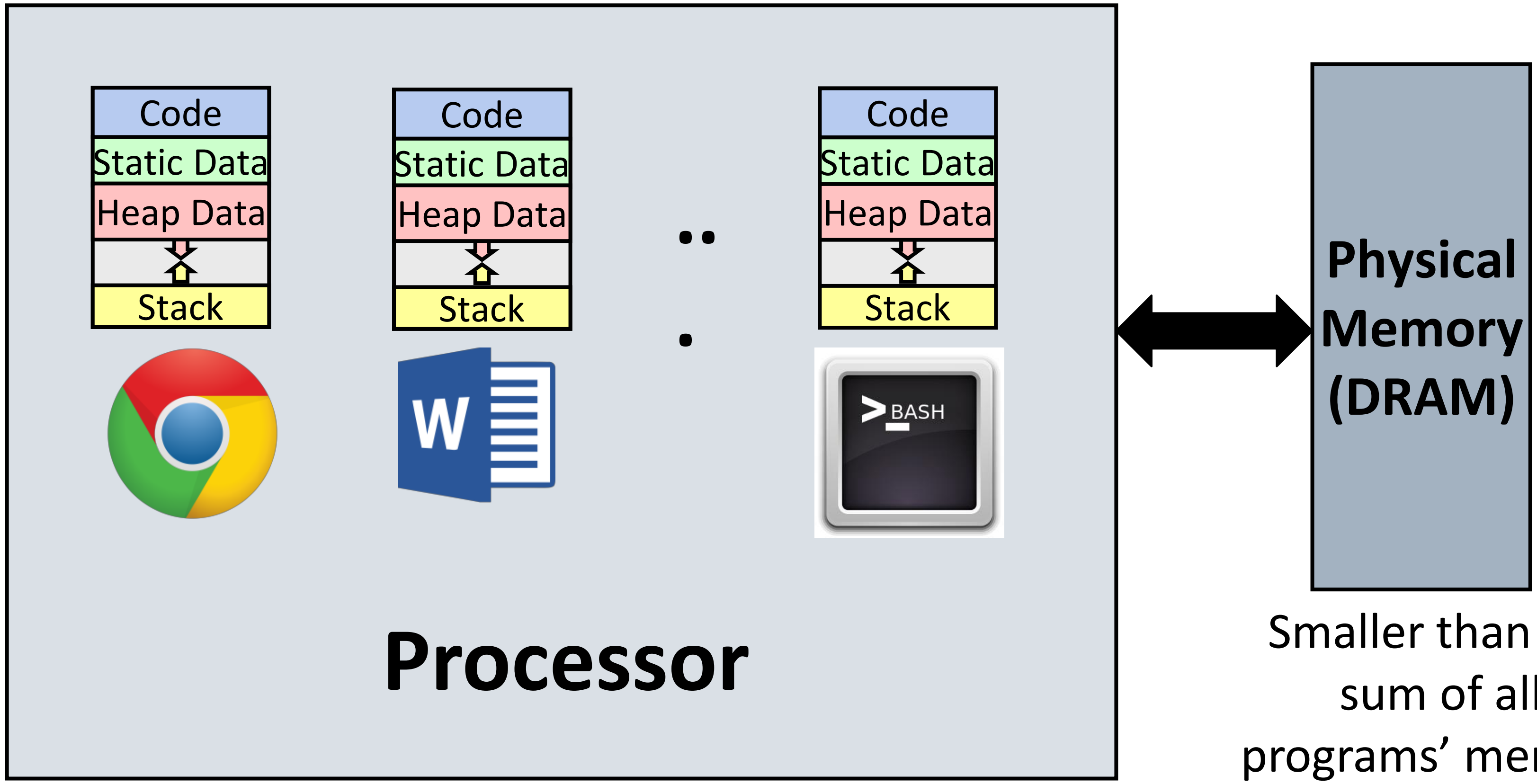
Memory: Two Issues

1. We've been working with the abstraction that all programs have full, private access to memory
   - But in practice, multiple programs run at the same time!



   - What happens if two programs try to write to the same memory address??

Revisit real system view—multitasking



Smaller than the sum of all programs’ memory

Memory: Two Issues

2. Even if only one program is running, modern computers have 48-64 bit address spaces!
   - No computer actually has 18 exabytes (18 billion GBs)
   - What if a computer tries to write to address 0xFFFF…FFFF
   - Should it just crash??

Memory: Two Issues

- Modern systems use the same solution for both problems: **virtual memory**
  - In a nutshell, each program thinks it has full, private access to memory (it can safely index any address from 0x0-0xFFF…FFFF)
  - Hardware and software transparently maps these addresses to distinct addresses in DRAM and in hard disk / SSD
  - Focus for the next 3 lectures

Agenda

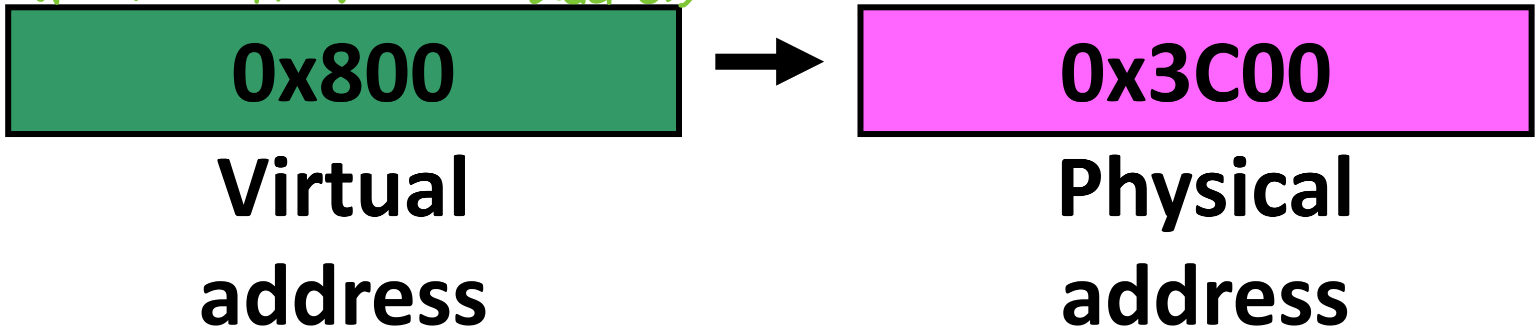
- Motivation
- **Virtual Memory Principles**
- Page Tables
- Class Problem

Basics of Virtual Memory

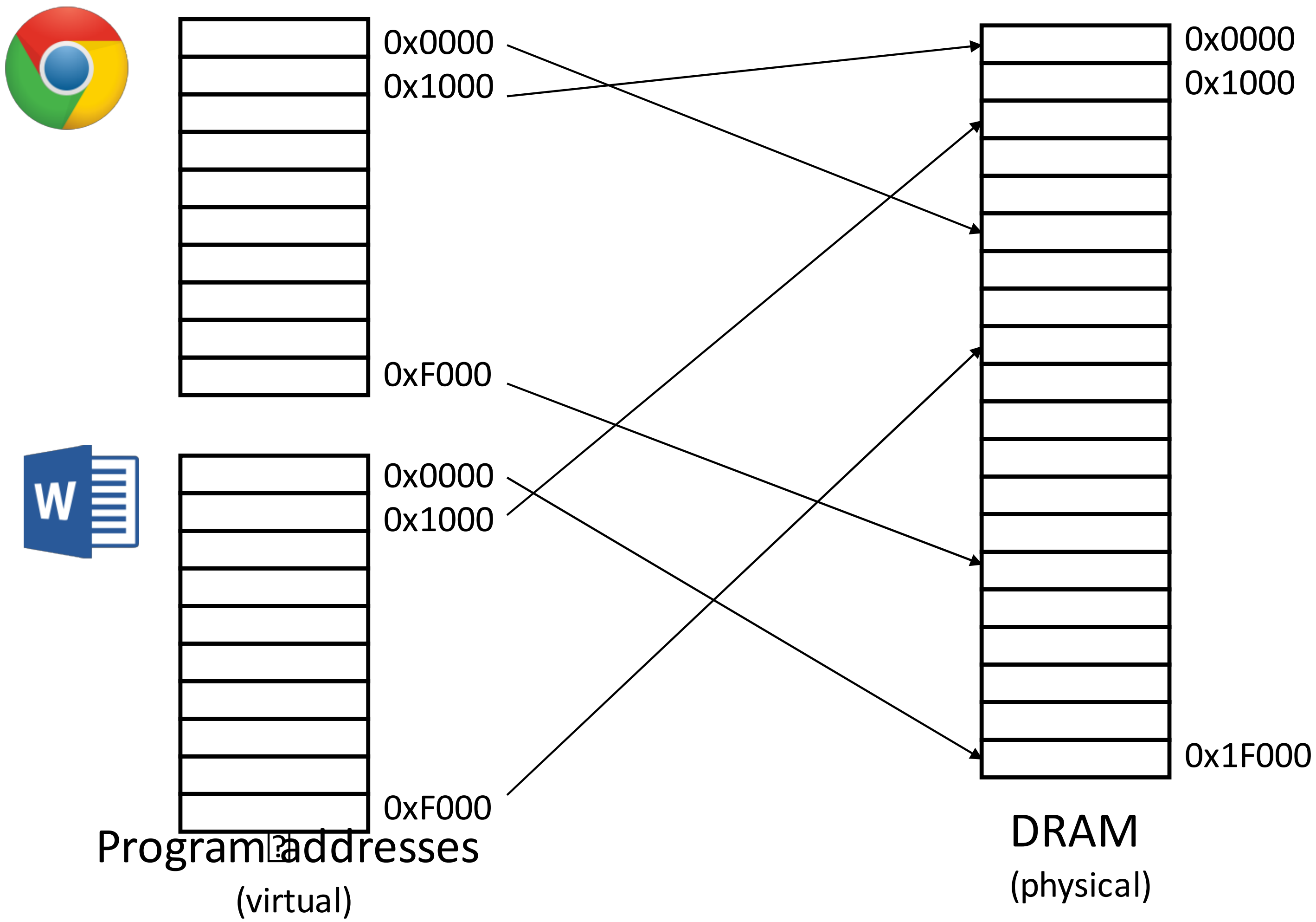
- Any time you see the word virtual in computer science and architecture it means "using a level of indirection".
  - Examples?

- Virtual memory hardware changes the virtual address the programmer sees into the physical one the memory chips see.

之前学的都是 virtual address



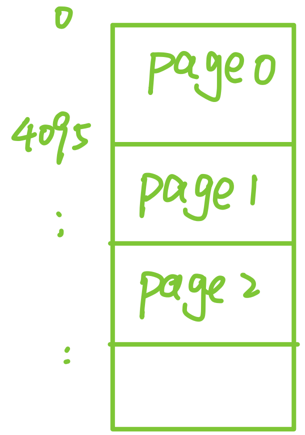
Virtual Memory



How to Translate Addresses?

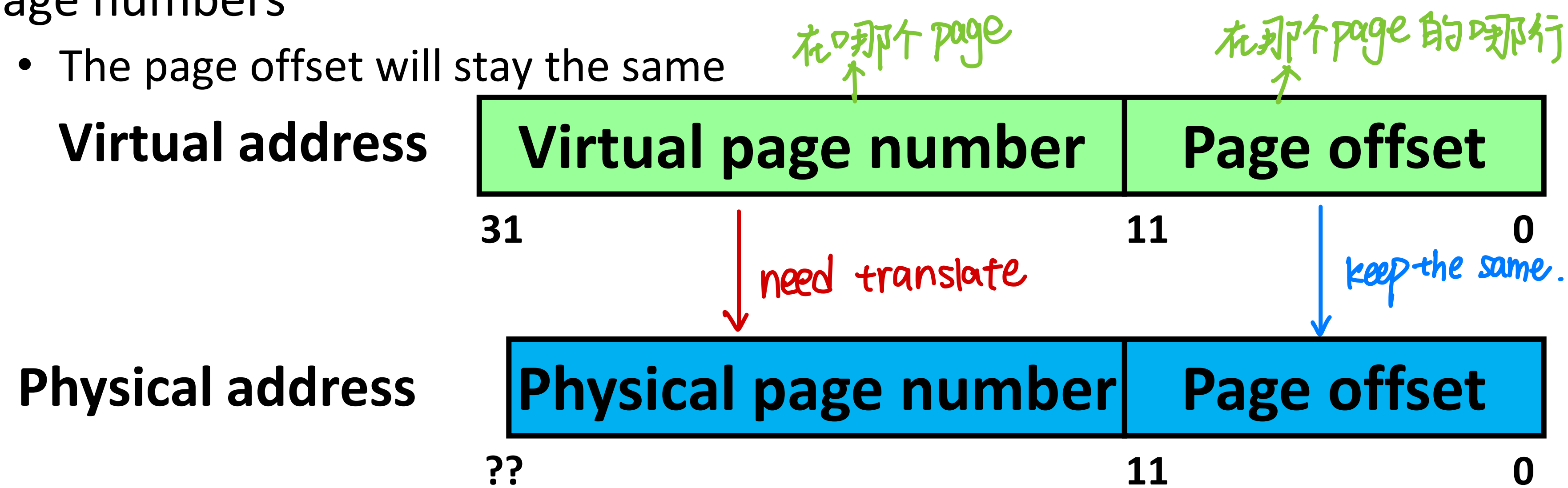
- Address Translation is not done entirely in hardware
- We'll get help in software via the **operating system**
- The operating system is a special set of programs
  - Always running (after the system boots)
  - Is in charge of **managing the hardware resources** for all other running programs
  - E.g. initializing memory for a starting program, managing the file system, choosing when a particular program gets to run…
  - … and translating virtual addresses into physical addresses!
- OS handles address translation by maintaining a data structure in main memory: **the page table**

Virtual memory terminology

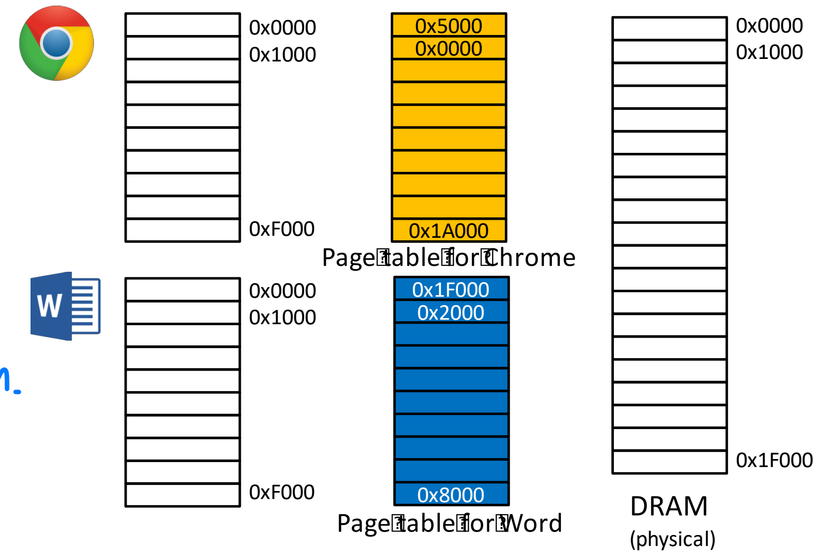


virtual memory 和 physical memory
都会被分成很多的 page.

- Memory is divided into fixed-size chunks called **Pages** (e.g., 4KB for x86)
  - Size of physical page = size of virtual page
  - A virtual address consists of
    - A virtual page number
    - A page offset field (low order bits of the address)
  - Translating a virtual address into a physical address only requires translating the page numbers
    - The page offset will stay the same



Page Table



This is a data structure that is maintained by the operating system.

- Translate page numbers using **page tables**
- Contains address translation information, i.e. virtual page # ➔ physical page #

virtual page numbers   physical page numbers

- Each process has its own page table
  - Maintained by operating system (OS)
- Page tables themselves are kept in memory by OS, and OS knows the physical address of the page tables
  - No address translation is required by the OS for accessing the page tables

Poll: What is the cost of this scheme? (select all that apply)

a) Uses up more memory
b) Takes longer to do loads and stores
c) Fewer addresses accessible by program
d) None of the above

the addresses here used by program is virtual address.
we truely have fewer physical address.

Why Pages?

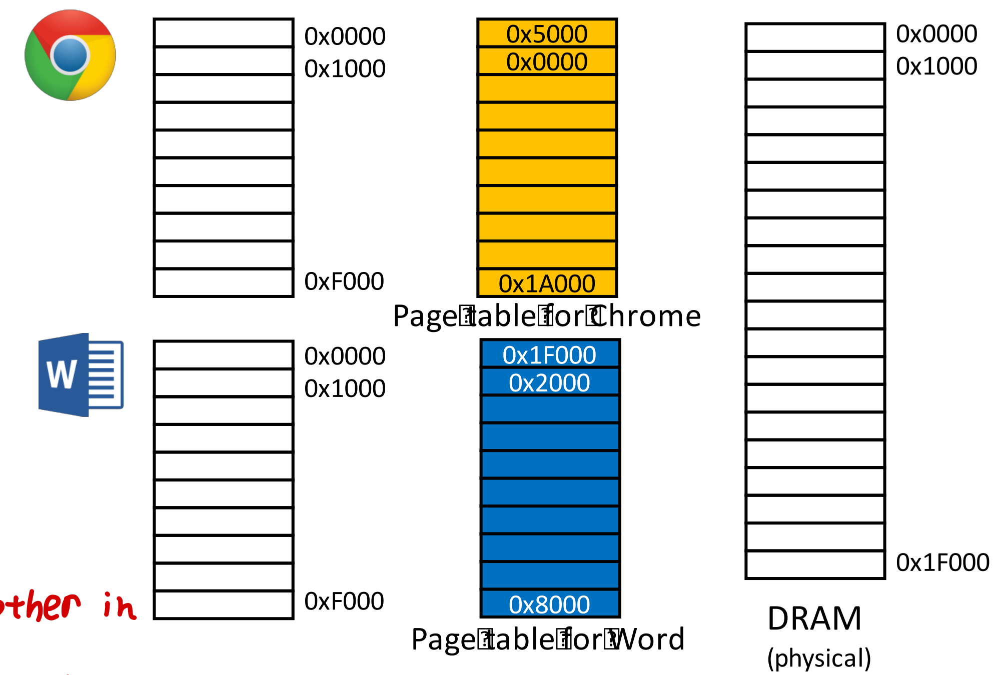
- Why have the idea of "pages"? Why not just have a mapping for each individual address?

  the larger each page is → the fewer pages there are in memory → the fewer entries we need in our page table.

  If I translate every address independently, I need to have an entry in the page table for every single address → fill up the entirety of memory just by keeping the translation information. no space to hold data.

  - Equivalent to asking: "why not have pages of size 1 byte?"
  - Otherwise - need a mapping entry for every single element of memory
  - The mapping data would take up as much space as the actual program data!
  - Also would screw up spatial locality of cache blocks (things contiguous in virtual memory wouldn't be contiguous in physical memory)

    If two addresses are nearby each other in virtual memory (as long as they are in the same page) → then they will be in the same distance from each other in the physical memory.



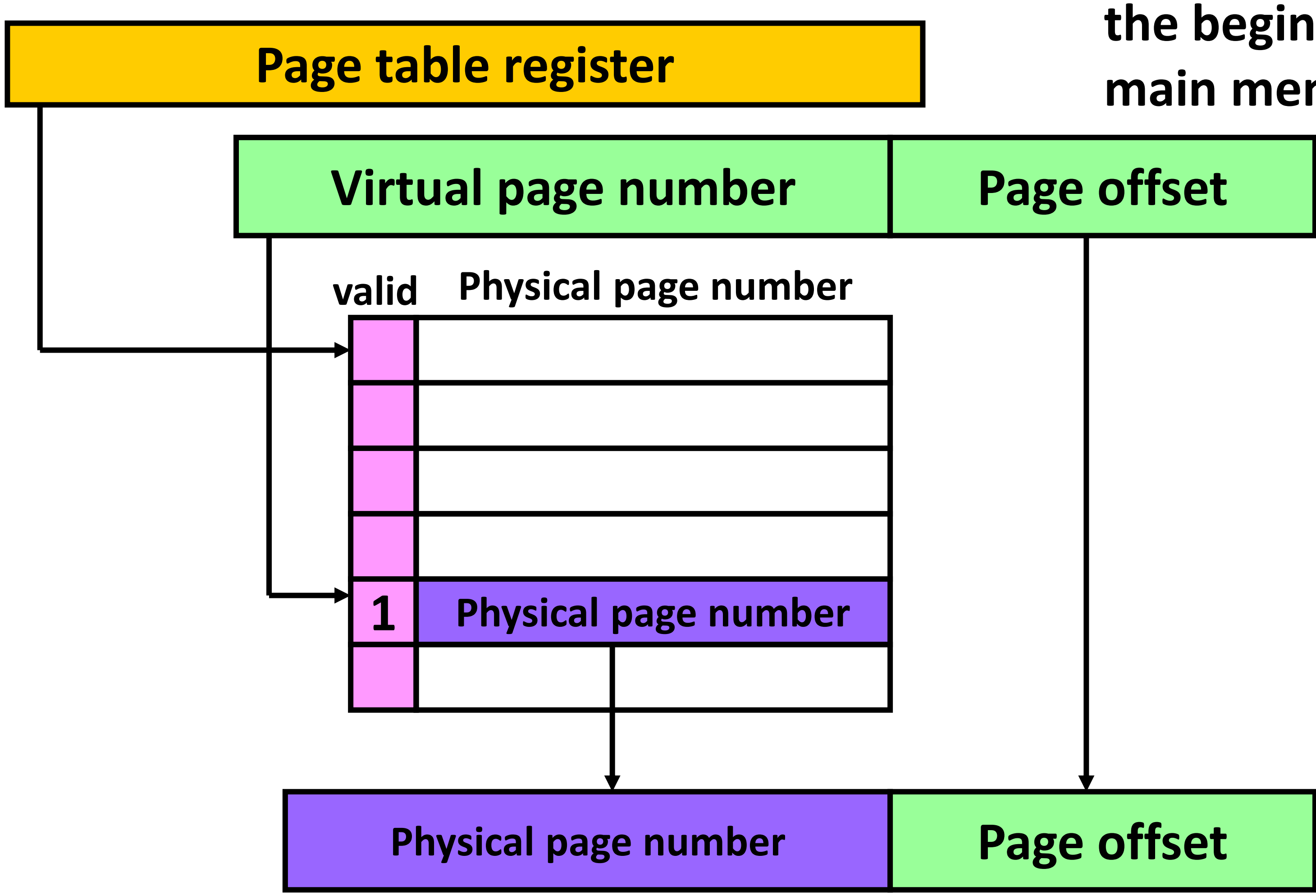
Agenda

- Motivation
- Virtual Memory Principles
- **Page Tables**
- Class Problem

• it's a data structure
• it's stored in regular DRAM.

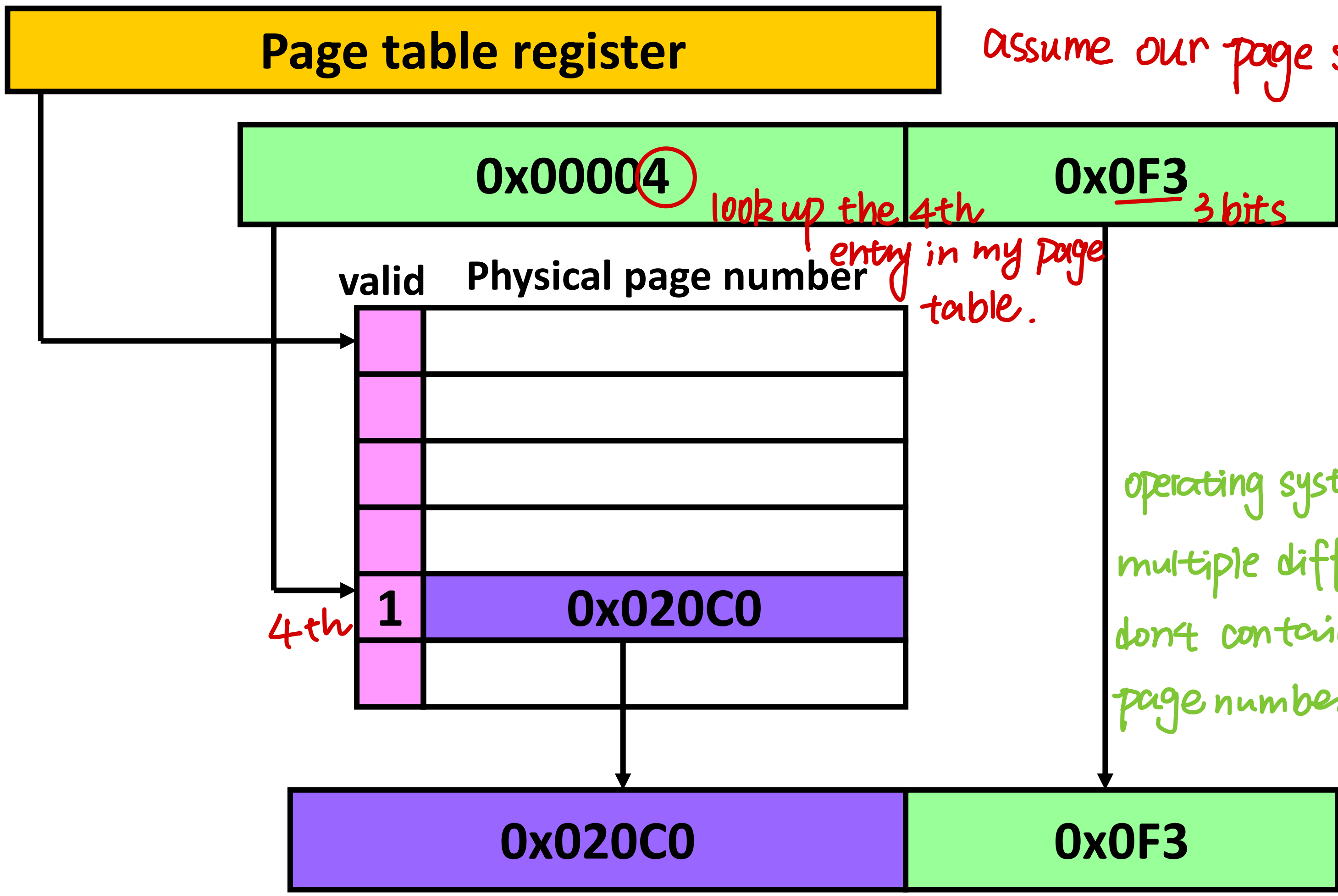
Page table components

The *Page table register* points to the beginning of the page table in main memory



Page table components - Example

Virtual address = 0x000040F3



Physical address = 0x020C00F3

assume our page size is : 4 KB.

$KB = 2^{10}$ $4 = 2^2$

$\underbrace{\quad}_{2^{12}}$

↓

3 Hex number.

operating system ensures that multiple different page tables don't contain the same physical page number cross different programe.

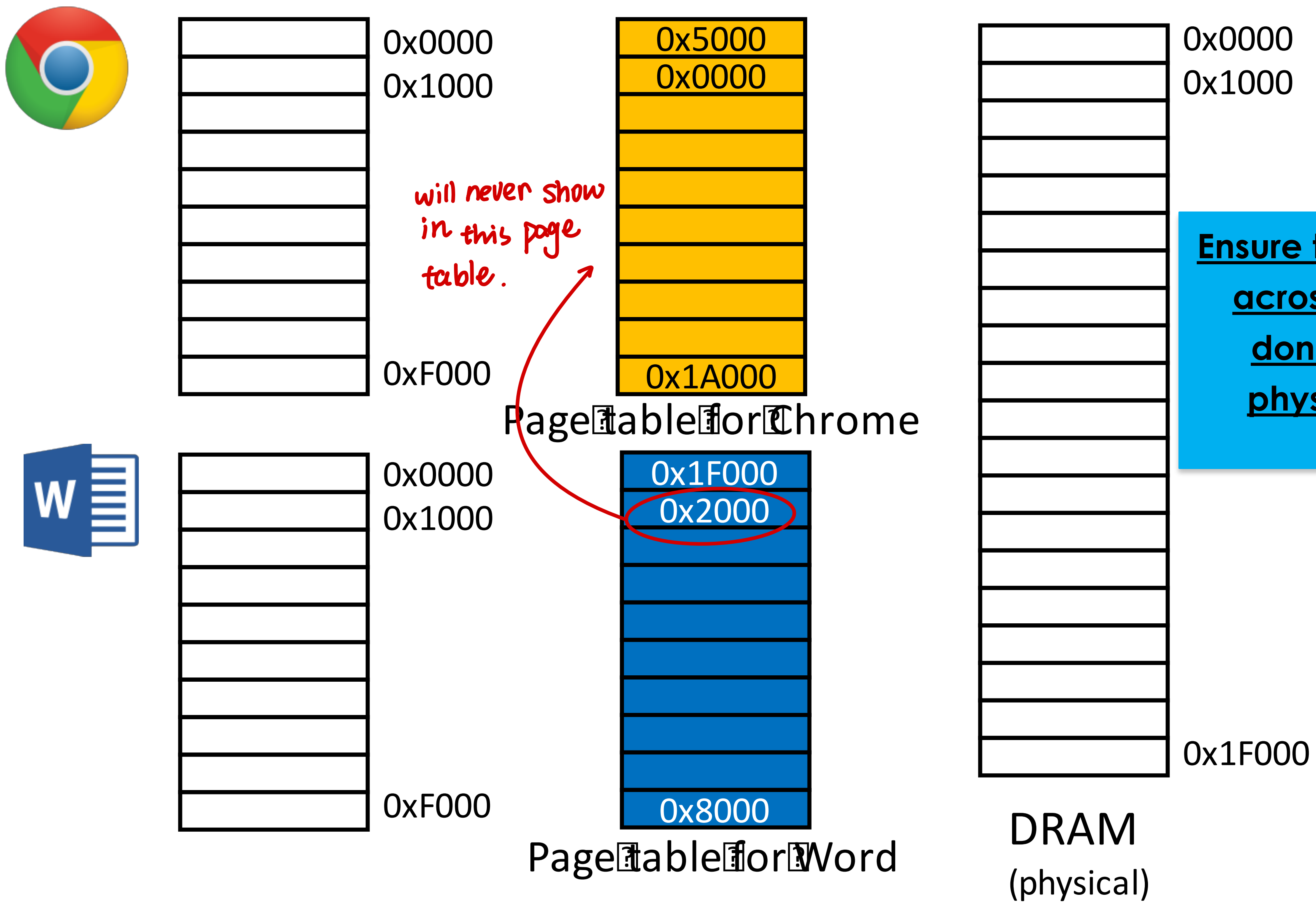
↓ then

we can make sure different programe will not access the same physical address.

Virtual Memory Goals

- VM should provide the following 3 capabilities to the programs:
  1. **Transparency**
     - Don't need to know how other programs are using memory
  2. **Protection**
     - No program can access the data of any other program
  3. **Programs not limited by DRAM capacity**
     - Each program can have more data than DRAM size

1-2: How to achieve transparency & protection?



Ensure that page table entries across different programs don't contain the same physical page numbers

3. How to be not limited by DRAM capacity?

- Use disk as temporary space in case memory capacity is exhausted
  - This temporary space in disk is called swap partition in Linux-based systems
  - For fun check swap space in a linux system by:

`$: top`

```
1. ssh
Tasks: 662 total,   1 running, 661 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.0 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  32704372 total, 10813444 used, 21890928 free,  1018840 buffers
KiB Swap: 35162108 total,    89248 used, 35072860 free.  7053764 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
60256 nehaag    20   0   25356   3356   2444 R   6.0  0.0   0:00.02 top
    1 root      20   0   38424   9040   2780 S   0.0  0.0   1:56.96 init
    2 root      20   0       0      0      0 S   0.0  0.0   0:02.21 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   6:20.75 ksoftirqd/0
```

2. How to be not limited by DRAM capacity?

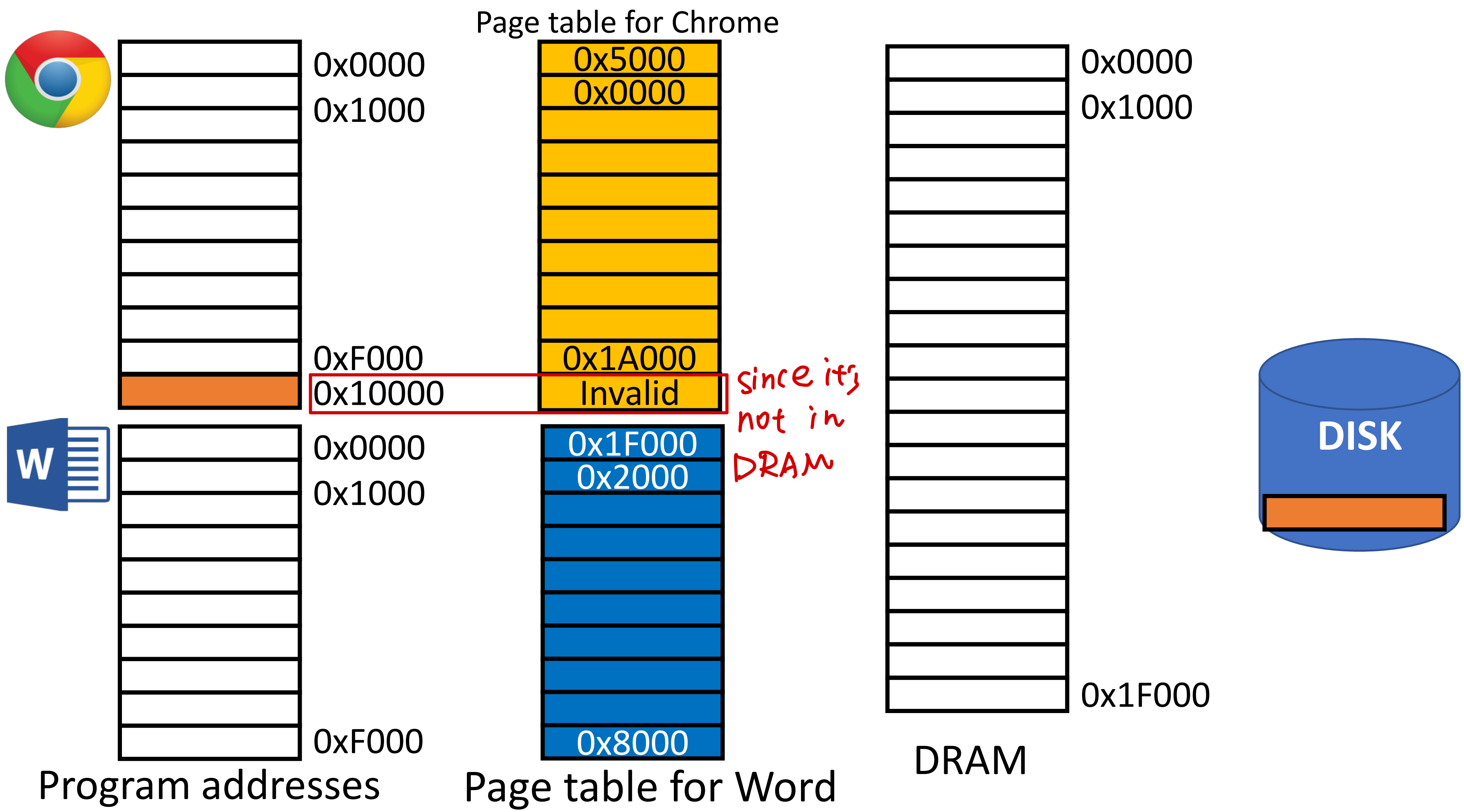
- We can mark a page table entry as "Invalid", indicating that the data for that page doesn't exist in main memory, but instead is located on the disk

  First time we try to access a new page, since it's Invalid, we won't be able to load it directly.

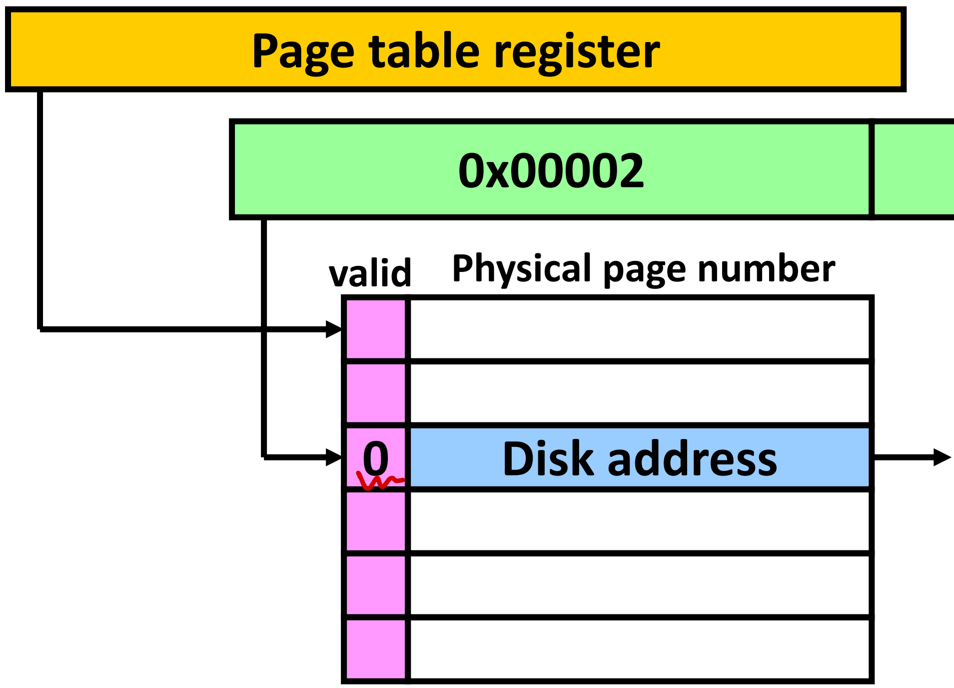
- Looking up a page table entry that corresponds to disk is called a **page fault**

  Any time that you look up somthing in the page table and it's marked as invalid. → then operating system will bring that data into memory for us.

2. How to be not limited by DRAM capacity?



Page faults



Exception: page fault

1. Stop this process
2. Pick page to replace
3. Write back data
4. Get referenced page
5. Update page table
6. Reschedule process

How do we find it on disk?

- That is not a hardware problem! Go take EECS 482! ☺
- This is the operating system's job. Most operating systems partition the disk into logical devices
  (C: , D: , /home, etc.)
- They also have a hidden partition to support the disk portion of virtual memory
  - Swap partition on UNIX machines
  - You then index into the correct page in the swap partition.

Agenda

- Motivation
- Virtual Memory Principles
- Page Tables
- **Class Problem**

Class Problem

4KB page size → 2^{12} 12 bits

Physical memory 16KB. → 4 pages.

virtual address : 20 bit.

8-bit	12-bit
page number.	page offset.

- Given the following:
  - 4KB page size, physical memory of 16KB, page table stored in physical page 0 and can never be evicted, 20 bit, byte-addressable virtual address space.
  - The page table initially has virtual page 0 in physical page 1, virtual page 1 in physical page 2 and no valid data in other physical pages.
- Fill in the table on the next slide for each reference
  - Note: like caches we'll use LRU when we need to replace a page.

Physical memory

PP0	Page0 → page table
PP1	VP0
PP2	~~VP1~~ VP30
PP3	~~VP20~~ VP01

Page table (2^8 entries)

0	1	1
1	x	a 1 ~~2~~ 3
20	x	0 ~~3~~
30	1	2

Class Problem (continued)

4KB page size, physical memory of 16KB, page table stored in physical page 0 and can never be evicted, 20 bit, byte-addressable virtual address space.

The page table initially has virtual page 0 in physical page 1, virtual page 1 in physical page 2 and no valid data in other physical pages.

Virt addr	Virt page	Page fault?	Phys addr
0x00F0C	0x00	N	0x01F0C
0x01F0C	0x01	N	0x02F0C
0x20F0C	0x20	Y	0x03F0C
0x00100	0x00	N	0x01100
0x00200	0x00	N	0x01200
0x30000	0x30	Y	0x02000
0x01FFF	0x01	Y	0x03FFF
0x00200	0x00	N	0x01200

page offset

Poll: How many hex digits should the page number be?

Class Problem (continued)

4KB page size, physical memory of 16KB, page table stored in physical page 0 and can never be evicted, 20 bit, byte-addressable virtual address space.

4 pages

Physical mem

PP0	PT
PP1	VP0
PP2	LRU → ~~VP1~~ VP30
PP3	~~VP20~~ VP1

Page Table (Hex)

The page table initially has virtual page 0 in physical page 1, virtual page 1 in physical page 2 and no valid data in other physical pages.

Virt addr	Virt page	Page fault?	Phys addr
page number / page offset			
0x00F0C	0x00	Not	0x01F0C
0x01F0C	0x01	Not	0x02F0C
0x20F0C	0x20	Yes	0x03F0C
0x00100	0x00	Not	0x01100
0x00200	0x00	Not	0x01200
0x30000	0x30	Yes	0x02000
0x01FFF	0x01	Yes.	0x03FFF
0x00200	0x00	No	0x01200

$\frac{2^{20}}{2^{12}} \rightarrow 2^8$ virtual pages.

Poll: How many hex digits should the page number be?

4KB

page offset : 12 bits.

↓

3 hex digits.

20 bits

Virtual page number	page offset.

look up in the page table and get corresponding physical page

12 bits. → 3 hex digits

8 bits → 2 hex digits.

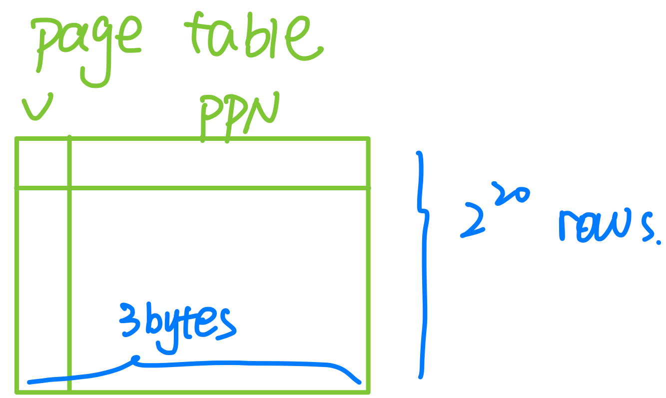
Class Problem (continued)

4KB page size, physical memory of 16KB, page table stored in physical page 0 and can never be evicted, 20 bit, byte-addressable virtual address space.

The page table initially has virtual page 0 in physical page 1, virtual page 1 in physical page 2 and no valid data in other physical pages.

Virt addr	Virt page	Page fault?	Phys addr
0x00F0C	0x0	N	0x1F0C
0x01F0C	0x1	N	0x2F0C
0x20F0C	0x20	Y (into 3)	0x3F0C
0x00100	0x0	N	0x1100
0x00200	0x0	N	0x1200
0x30000	0x30	Y (into 2)	0x2000
0x01FFF	0x1	Y (into 3)	0x3FFF
0x00200	0x0	N	0x1200

Size of the page table



- How big is a page table entry?
  - For 32-bit virtual address: $32 - 12 = 20$ bits → for virtual page number
    - If the machine can support 1GB = 2^{30} bytes of physical memory and we use pages of size 4KB = 2^{12},
    - then the physical page number is 30-12 = 18 bits. → 18 bits physical page number.
      Plus another valid bit + other useful stuff (read only, dirty, etc.)
    - Let say about 3 bytes.
- How many entries in the page table?
  - 1 entry per virtual page
  - ARM virtual address is 32 bits – 12 bit page offset = 20
  - Total number of virtual pages = 2^{20}

$2^{20} \times 3$ bytes ≈ 3MB.

- Total size of page table = Number of virtual pages

  * Size of each page table entry

  = $2^{20} \times 3$ bytes ~ 3 MB

when 64-bit system it will become bigger. But most of the row in the page table is invalid

Next lecture: improve

Size of the page table

$m \times w$

w

m

30

18 | 12
physical page number | page offset
physical memory.

- How big is a page table entry?
  - For 32-bit virtual address: $(2^{10})^3 = 2^{30}$ bytes.
    - If the machine can support 1GB = 2^{30} bytes of physical memory and we use pages of size 4KB = 2^{12},
    - then the physical page number is 30-12 = 18 bits.
      Plus another valid bit + other useful stuff (read only, dirty, etc.) 6.
    - n: Let say about 3 bytes. 一共 24 bits.
- How many entries in the page table? m的大小取决于 virtual address 的大小.
  - 1 entry per virtual page
  - ARM virtual address is 32 bits – 12 bit page offset = 20
  - Total number of virtual pages = 2^{20}

32-bits virtual address.

20 | 12
virtual page number | page offset

- Total size of page table = Number of virtual pages

  MB $= 2^{20}$ bytes

  * Size of each page table entry 32

  $m = 2^{20}$.

  = 2^{20} x 3 bytes ~ 3 MB

Total. $m \times n = 2^{20} \times 3$ bytes $\approx$ 3MB.

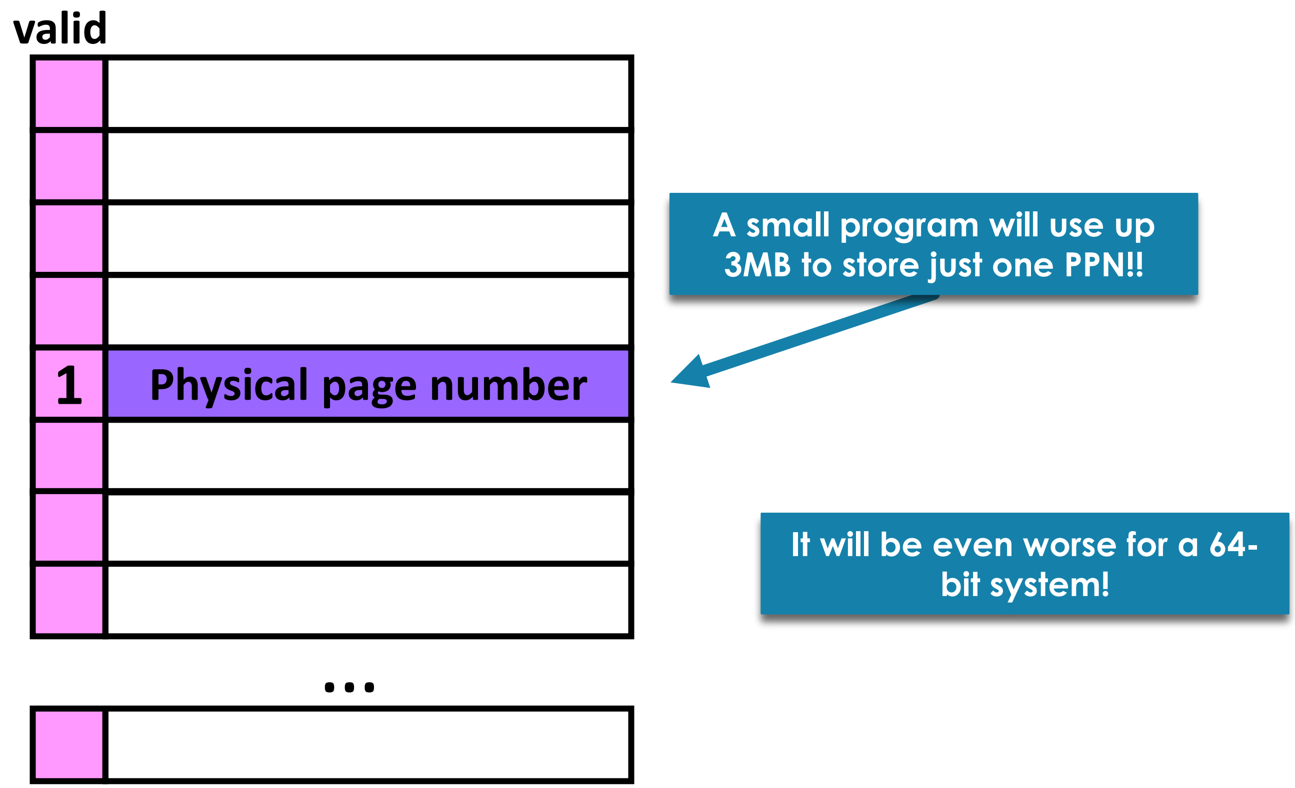
Next time

- Improving Virtual Memory Design
  - Multi-level page-tables

Extra Slides

How can you organize the page table?

1. Single-level page table occupies continuous region in physical memory
   - Previous example always takes 3MB regardless of how much virtual memory is used



How can you organize the page table?

2. Option 2: Use a multi-level page table
   - 1^{st} level page table (much smaller!) holds addresses 2^{nd} level page tables
     - 2^{nd} level page tables hold translation info, or 3^{rd} level page tables if we wanna go deeper
     - Only allocate space for 2^{nd} level page tables that are used

valid	PPN
1	0x1
1	0x2
1	0x3

Single-level: Tons of wasted space!

valid	2^{nd} level page table
1	0x1000

valid	PPN
1	0x1
1	0x2
1	0x3

no limitation to make it deeper, but it will cause longer latency. But the space for each table gets smaller so, it's a trade off.

Multi-Level Page Table

- Only allocate second (and later) page tables when needed
- Program starts: everything is invalid, only first level is allocated

valid	2nd level page table
0	
0	
0	
0	

- As we access more, second level page tables are allocated

Multi-level: Size is proportional to amount of memory used

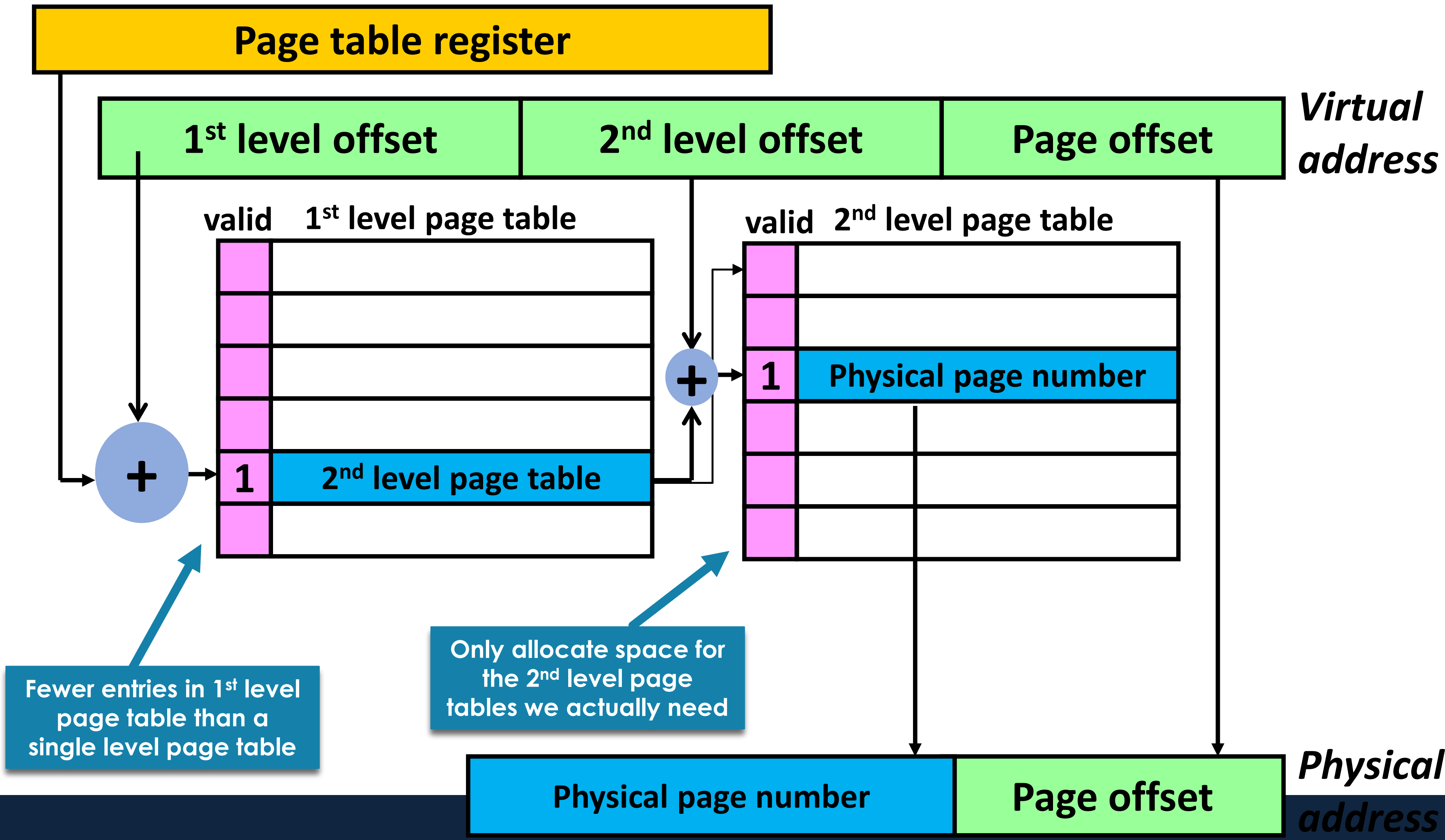
valid	2nd level page table
1	0x1000
1	0x3500

valid	PPN
1	0x1
1	0x6
1	0x2
1	0x1f

valid	PPN
1	0x9a
1	0x3
0	
1	0xff

<u>Common case</u>: most programs use small portion of virtual memory space

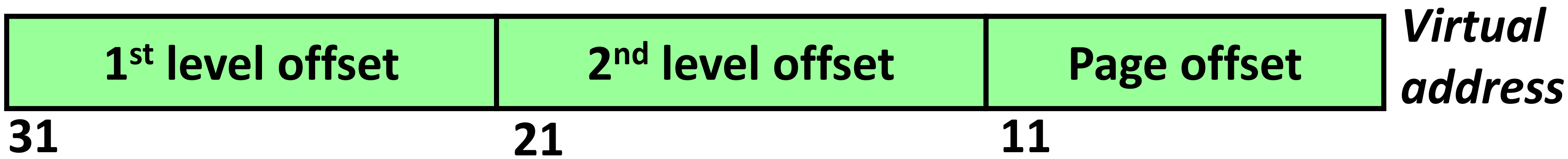
Hierarchical page table



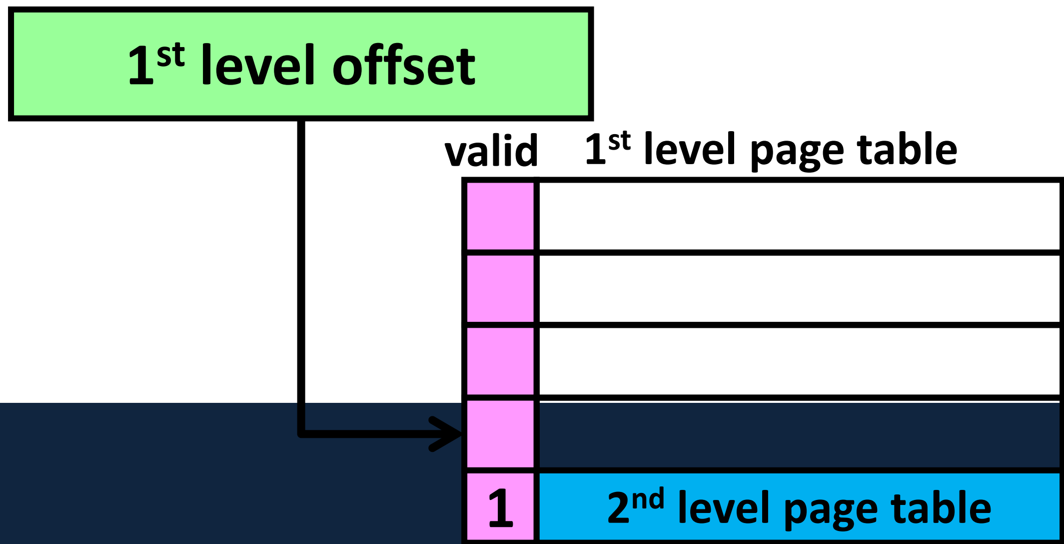
Agenda

- Motivation for Multi-level Page Tables
- **Example architecture**
- Class Problem: 32bit Intel x86
- Class Problem: Multi-Level VM Design
- VM Miscellanea

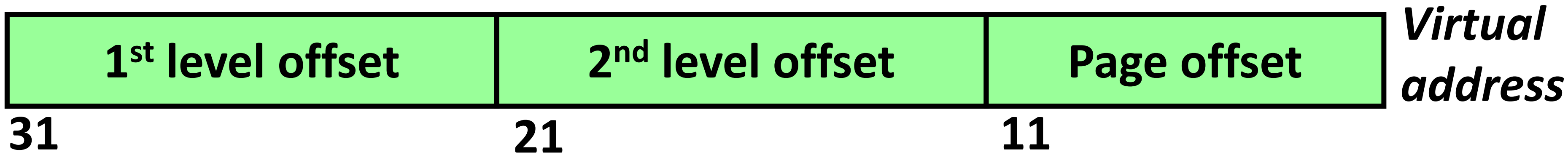
Hierarchical page table – 32bit Intel x86



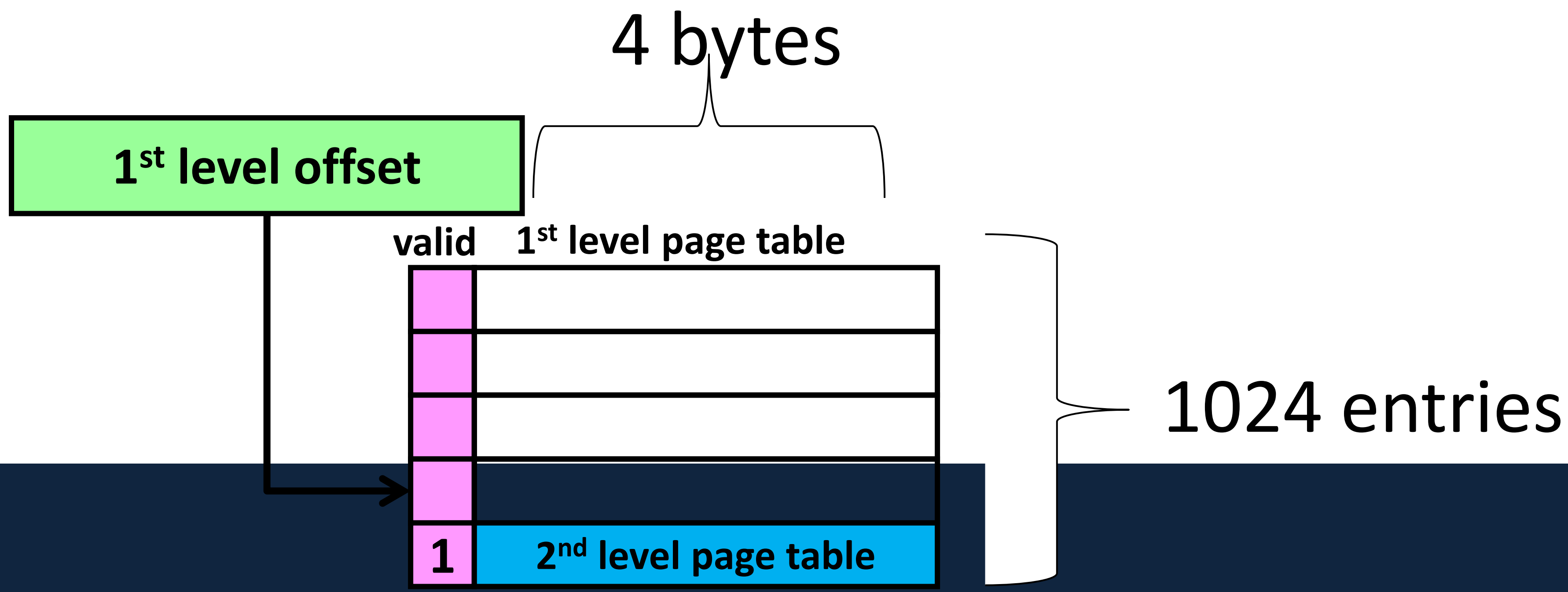
- How many bits in the virtual 1st level offset field? 10
- How many bits in the virtual 2nd level offset field? 10
- How many bits in the page offset? 12
- How many entries in the 1st level page table? $2^{10}=1024$



Hierarchical page table – 32bit Intel x86

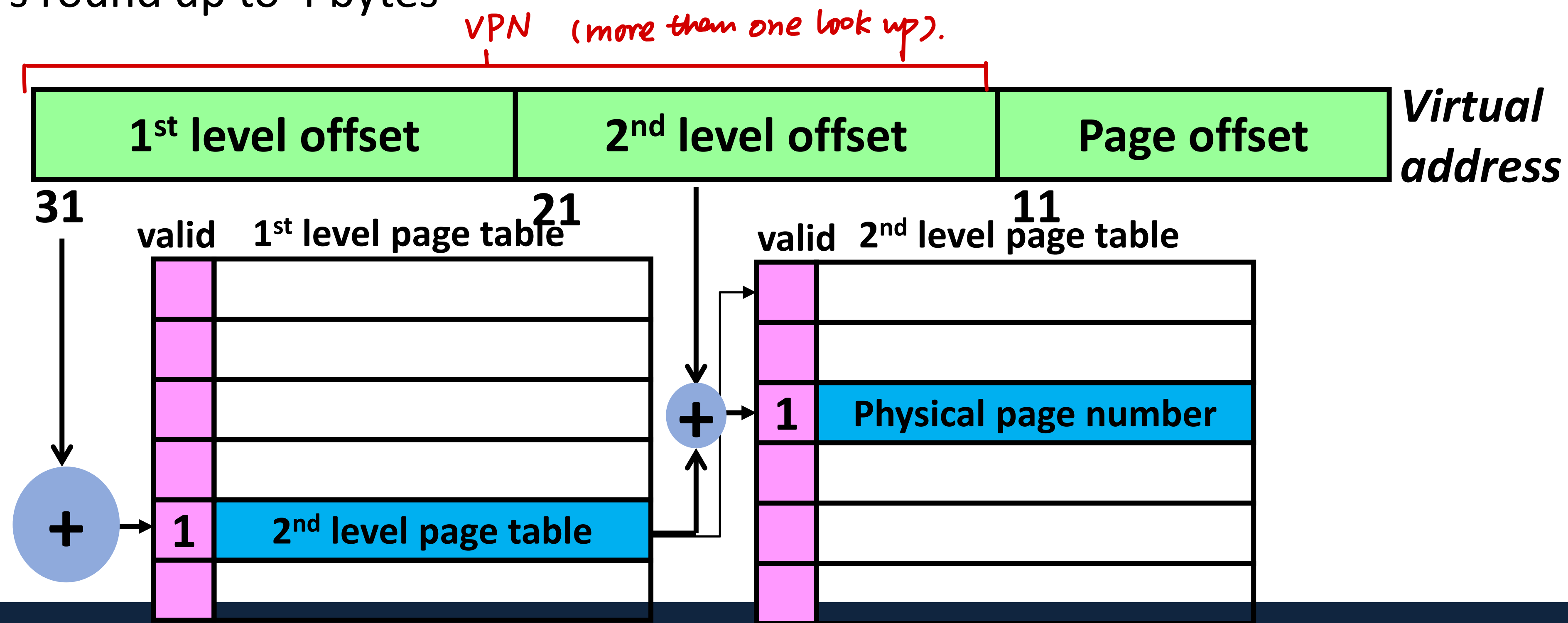


- Let's say physical address size + overhead bits is 4 bytes per entry
- Total size of 1st level page table
  - 4 bytes * 1024 entries = 4 KB



Hierarchical page table

- How many entries in the 2nd level of the page table?
  - $2^{10} = 1024$
- How many bytes for each VPN in a 2nd level table?
  - Let's round up to 4 bytes



Hierarchical page table – 32bit Intel x86

1st level offset	2nd level offset	Page offset
31	21	11

Virtual address

• How many bits in the virtual 1st level offset field?	10
• How many bits in the virtual 2nd level offset field?	10
• How many bits in the page offset?	12
• How many entries in the 1st level page table?	2^{10}=1024
• How many bytes for each entry in the 1st level page table?	4
• How many entries in the 2nd level of the page table?	2^{10}=1024
• How many bytes for each entry in a 2nd level table?	~4
• **What is the total size of the page table?**	**4K+n*4K**

(here *n* is number of valid entries in the 1st level page table)

n should be proportional to the amount of memory that is being used in this program.

不像1层page table是定值.